



# Funções Hash

**Prof. Dr. Iaçanã Ianiski Weber**

*Confiabilidade e Segurança de Software*

98G08-4

*Agradecimentos especiais ao Prof. Avelino Zorzo e aos Autores Christof Paar e Jan Pelzl pelo material.*

# Índice

1 Porque Precisamos de Funções Hash

2 Propriedades de Funções de Hash

3 SHA-3

1 Porque Precisamos de Funções Hash

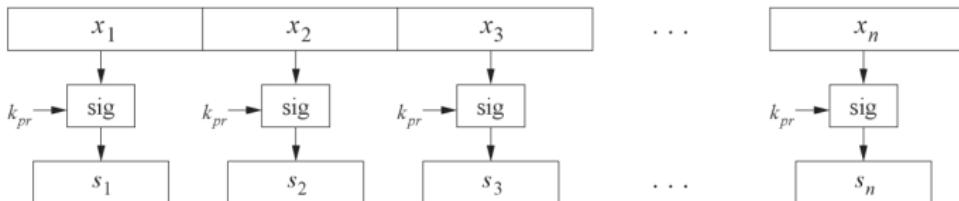
2 Propriedades de Funções de Hash

3 SHA-3

# Por que usar Funções de Hash para Assinaturas?

## O Problema: Assinatura ingênua de mensagens longas

Assinar uma mensagem longa em blocos separados gera uma assinatura de tamanho similar ao da própria mensagem, o que é ineficiente.



Isto gera três problemas principais:

- **Sobrecarga Computacional:** Operações de chave pública são lentas e custosas para cada bloco.
- **Sobrecarga de Mensagem:** A assinatura final fica muito grande.
- **Limitações de Segurança:** Um atacante pode reordenar, remover ou duplicar blocos para forjar novas mensagens.

## Solução

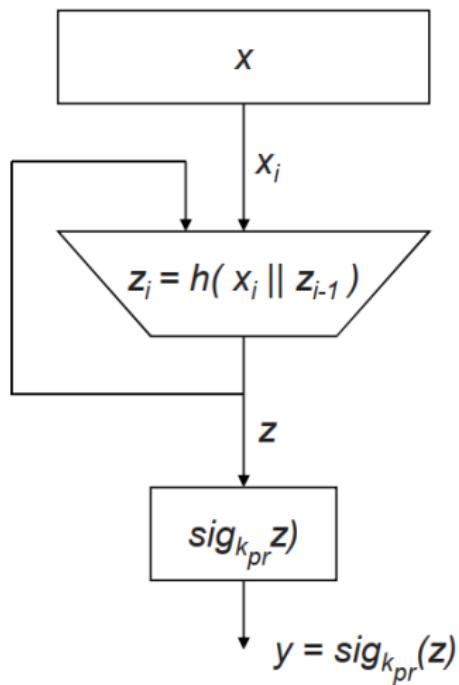
Em vez de assinar a mensagem inteira, assina-se apenas seu **digest** (ou **hash**).

→ Igualmente seguro, muito mais rápido.

## Necessidade

Isso requer o uso de **Funções de Hash criptográficas**.

# Assinatura Digital com Função Hash



## Notes:

- $x$  has fixed length
- $z, y$  have fixed length
- $z, x$  do not have equal length in general
- $h(x)$  does not require a key.
- $h(x)$  is public.

# Protocolo Básico de Assinatura Digital usando Função Hash

Alice

Bob

 $K_{pub}$ 

$$z = h(x)$$

$$s = \text{sig}_{K_{pr}}(z)$$

 $(x, s)$ 

$$z' = h(x)$$

$$\text{ver}_{K_{pub}}(s, z') = \text{true/false}$$

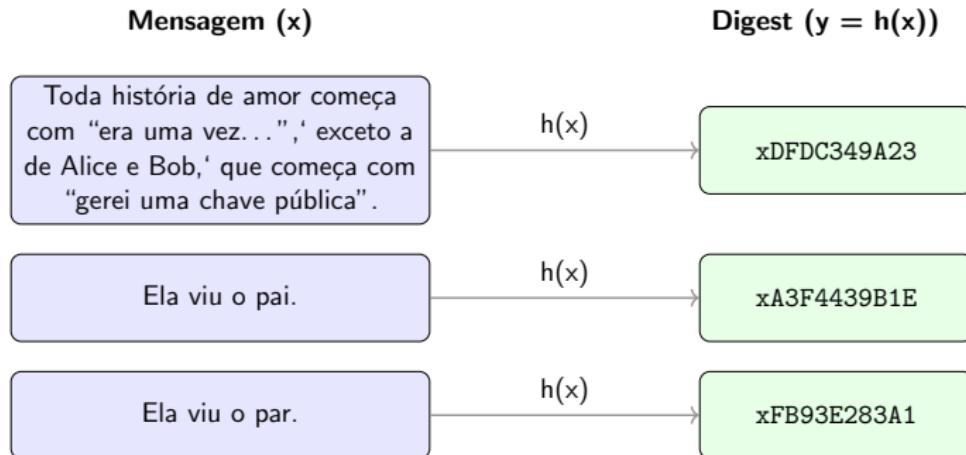
# Índice

1 Porque Precisamos de Funções Hash

2 Propriedades de Funções de Hash

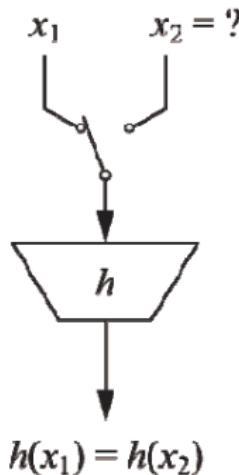
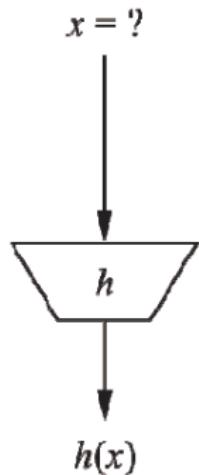
3 SHA-3

# O que uma Função de Hash faz?



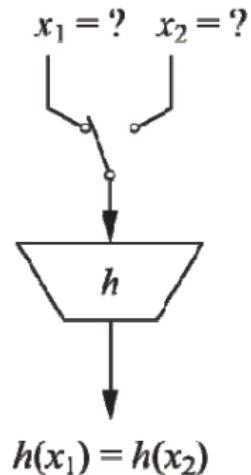
- **Entrada de tamanho arbitrário:** A função  $h$  opera sobre uma mensagem  $x$  de qualquer comprimento.
- **Saída de tamanho fixo:** O resultado (digest)  $y = h(x)$  tem sempre o mesmo comprimento (ex: 256 bits).
- **Função de mão única (one-way):** Dado  $x$ , é fácil computar  $h(x)$ . Dado  $h(x)$ , é computacionalmente inviável encontrar  $x$ .

# As três propriedades das Funções Hash



preimage resistance

second preimage  
resistance



collision resistance

# As Três Garantias Fundamentais

- **Resistência à pré-imagem (Preimage resistance)**

Para uma dada saída  $z$ , é computacionalmente inviável encontrar qualquer entrada  $x$  tal que  $h(x) = z$ .

- Essencialmente, a propriedade de ser uma função de **mão única (one-way)**.

- **Resistência à segunda pré-imagem (Second preimage resistance)**

Dado um bloco de dados  $x_1$ , é computacionalmente inviável encontrar *outro* bloco  $x_2$  tal que  $x_1 \neq x_2$  e que  $h(x_1) = h(x_2)$ .

- Protege contra a falsificação de um documento já assinado.

- **Resistência à colisão (Collision resistance)**

É computacionalmente inviável encontrar *qualquer par* de entradas distintas  $x_1, x_2$  tal que  $x_1 \neq x_2$  e que  $h(x_1) = h(x_2)$ .

- É uma propriedade mais forte que a resistência à segunda pré-imagem.

# O Paradoxo do Aniversário

- A **resistência à colisão** é a propriedade que, na prática, causa mais problemas de segurança.

# O Paradoxo do Aniversário

- A **resistência à colisão** é a propriedade que, na prática, causa mais problemas de segurança.
- **Pergunta análoga:** Quantas pessoas são necessárias em uma sala para que a probabilidade de duas delas fazerem aniversário no mesmo dia seja de 50%?

# O Paradoxo do Aniversário

- A **resistência à colisão** é a propriedade que, na prática, causa mais problemas de segurança.
- **Pergunta análoga:** Quantas pessoas são necessárias em uma sala para que a probabilidade de duas delas fazerem aniversário no mesmo dia seja de 50%?
  - A resposta intuitiva seria  $365/2 \approx 183$ ?

# O Paradoxo do Aniversário

- A **resistência à colisão** é a propriedade que, na prática, causa mais problemas de segurança.
- **Pergunta análoga:** Quantas pessoas são necessárias em uma sala para que a probabilidade de duas delas fazerem aniversário no mesmo dia seja de 50%?
  - A resposta intuitiva seria  $365/2 \approx 183$ ? **Não!**
- Apenas **23 pessoas** são suficientes! Isso é conhecido como o **Paradoxo do Aniversário**.

# O Paradoxo do Aniversário

- A **resistência à colisão** é a propriedade que, na prática, causa mais problemas de segurança.
- **Pergunta análoga:** Quantas pessoas são necessárias em uma sala para que a probabilidade de duas delas fazerem aniversário no mesmo dia seja de 50%?
  - A resposta intuitiva seria  $365/2 \approx 183$ ? **Não!**
- Apenas **23 pessoas** são suficientes! Isso é conhecido como o **Paradoxo do Aniversário**.
- **Implicação para Hashes:** Um ataque para encontrar colisões (chamado de *birthday attack*) é muito mais eficiente do que um ataque de força bruta. A busca leva aproximadamente  $2^{n/2}$  passos para uma função de hash com saída de  $n$  bits.

# O Paradoxo do Aniversário

- A **resistência à colisão** é a propriedade que, na prática, causa mais problemas de segurança.
- **Pergunta análoga:** Quantas pessoas são necessárias em uma sala para que a probabilidade de duas delas fazerem aniversário no mesmo dia seja de 50%?
  - A resposta intuitiva seria  $365/2 \approx 183$ ? **Não!**
- Apenas **23 pessoas** são suficientes! Isso é conhecido como o **Paradoxo do Aniversário**.
- **Implicação para Hashes:** Um ataque para encontrar colisões (chamado de *birthday attack*) é muito mais eficiente do que um ataque de força bruta. A busca leva aproximadamente  $2^{n/2}$  passos para uma função de hash com saída de  $n$  bits.

# O Paradoxo do Aniversário

## Por que o paradoxo funciona?

- A intuição falha porque não consideramos o número de **pares** possíveis. Com apenas 23 pessoas, existem 253 pares distintos, aumentando drasticamente a chance de uma coincidência.
- Em um ataque, o atacante não precisa quebrar o hash de um documento específico. Ele pode gerar várias versões de um documento bom e de um malicioso até que *qualquer par* entre eles produza o mesmo hash, o que é um problema muito mais fácil de resolver.

# O Paradoxo do Aniversário

## Por que o paradoxo funciona?

- A intuição falha porque não consideramos o número de **pares** possíveis. Com apenas 23 pessoas, existem 253 pares distintos, aumentando drasticamente a chance de uma coincidência.
- Em um ataque, o atacante não precisa quebrar o hash de um documento específico. Ele pode gerar várias versões de um documento bom e de um malicioso até que *qualquer par* entre eles produza o mesmo hash, o que é um problema muito mais fácil de resolver.

**Recomendação de Segurança:** Para se proteger deste ataque, a recomendação é que a saída de uma função de hash deve ter **no mínimo 224 bits**.

# O Ataque de Aniversário e a Segurança Moderna

- O **Paradoxo do Aniversário** mostra que é muito mais fácil encontrar *qualquer* colisão do que uma específica.
  - Para um hash de  $n$  bits, a complexidade de um ataque de aniversário é de aproximadamente  $2^{n/2}$  operações.
- **O Padrão Antigo (Obsoleto)**
  - A recomendação de 160 bits (do algoritmo SHA-1) oferecia  $160/2 = 80$  bits de segurança contra colisão.
  - Este nível de segurança ( $2^{80}$ ) **não é mais considerado seguro** contra adversários com recursos significativos.
- **A Recomendação Atual (NIST)**
  - O NIST exige um nível de segurança de no mínimo **112 bits** contra colisão.
  - Para isso, a saída do hash deve ter no mínimo  $2 \times 112 = 224$  bits.
- **Conclusão Prática para Novos Sistemas:**

# O Ataque de Aniversário e a Segurança Moderna

- O **Paradoxo do Aniversário** mostra que é muito mais fácil encontrar *qualquer* colisão do que uma específica.
  - Para um hash de  $n$  bits, a complexidade de um ataque de aniversário é de aproximadamente  $2^{n/2}$  operações.
- **O Padrão Antigo (Obsoleto)**
  - A recomendação de 160 bits (do algoritmo SHA-1) oferecia  $160/2 = 80$  bits de segurança contra colisão.
  - Este nível de segurança ( $2^{80}$ ) **não é mais considerado seguro** contra adversários com recursos significativos.
- **A Recomendação Atual (NIST)**
  - O NIST exige um nível de segurança de no mínimo **112 bits** contra colisão.
  - Para isso, a saída do hash deve ter no mínimo  $2 \times 112 = 224$  bits.
- **Conclusão Prática para Novos Sistemas:**
  - Use no mínimo **SHA-256** (saída de 256 bits).
  - Abandone completamente o uso de SHA-1.

# Funções de Hash: Lições Aprendidas (Versão Atualizada)

## • Natureza e Uso Principal

- Funções de Hash não usam chaves (*keyless*).
- Suas aplicações mais importantes são em assinaturas digitais e em códigos de autenticação de mensagens (MACs), como o HMAC.

## • Requisitos de Segurança Fundamentais

- As três propriedades de segurança essenciais são a resistência à pré-imagem (one-wayness), à segunda pré-imagem e à colisão.

## • Estado Atual dos Algoritmos

- **MD5 e SHA-1:** São considerados **inseguros**. O SHA-1 possui falhas graves e seu uso deve ser completamente descontinuado.
- **SHA-2:** A família de algoritmos SHA-2 (ex: SHA-256, SHA-512) continua sendo considerada segura e é o padrão de mercado atual.
- **SHA-3 (Atualização):** A competição SHA-3 **terminou em 2012**. O algoritmo vencedor (Keccak) foi padronizado em 2015. Ele serve como uma alternativa moderna ao SHA-2.

## • Recomendação de Tamanho (Atualizada)

- Para segurança de longo prazo e para resistir a ataques de colisão (ataques de aniversário), a recomendação atual é usar saídas de no mínimo **256 bits**.

# Índice

1 Porque Precisamos de Funções Hash

2 Propriedades de Funções de Hash

3 SHA-3

# SHA-3: Motivação e Contexto

- Em 2007, o NIST lança uma competição pública para definir uma nova função de hash: **SHA-3**.
- Motivação principal:
  - Ter uma alternativa a **SHA-2** com **projeto interno diferente**.
  - Garantir continuidade da segurança caso ocorra algum avanço criptanalítico contra o SHA-2.
- Diferentemente do AES (que substituiu o DES), **SHA-2 e SHA-3 foram projetados para coexistir**.
- O processo de seleção foi semelhante ao do AES:
  - Competição aberta.
  - Forte participação e escrutínio da comunidade científica.
- SHA-3 foi padronizado em 2015 como **FIPS 202**.

# Linha do Tempo da Competição SHA-3

- **Nov 2007:** NIST anuncia a chamada de algoritmos para SHA-3.
  - **Dez 2008:** 51 algoritmos selecionados para a **Rodada 1**.
  - **Jul 2009:** Redução para **14 algoritmos** na **Rodada 2**.
  - **Dez 2010:** NIST anuncia os **5 finalistas** (Rodada 3):
    - BLAKE, Grøstl, JH, Keccak e Skein.
  - **Out 2012:** **Keccak** é selecionado como base do SHA-3.
  - **Ago 2015:** Publicação oficial do **FIPS 202** (padronização do SHA-3).
- 
- “Padronizado” aqui significa tornar-se um **FIPS** (*Federal Information Processing Standard*) nos EUA.
  - Embora o uso obrigatório seja para sistemas governamentais dos EUA, a participação internacional no processo faz com que SHA-3 seja amplamente adotado globalmente, assim como o AES.

# Parâmetros de Segurança e SHAKE (XOFs)

- **Comprimentos de saída** suportados por SHA-3:

$$n \in \{224, 256, 384, 512\} \text{ bits}$$

- Pela **paradoxo do aniversário**, a complexidade de ataque por colisão é aproximadamente:

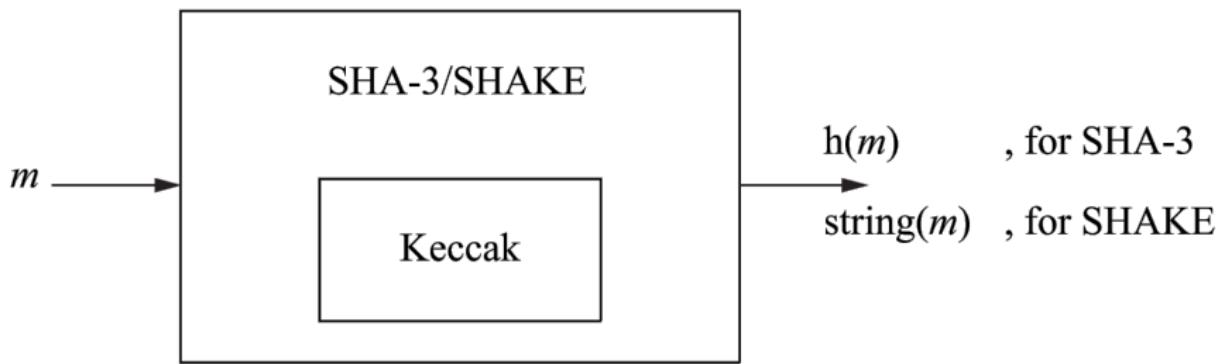
$$2^{112}, 2^{128}, 2^{192}, 2^{256}$$

para saídas de 224, 256, 384 e 512 bits, respectivamente.

- Esses níveis de segurança se alinham com:
  - **AES-128/192/256** (força contra força bruta),
  - **3DES** (força  $\approx 2^{112}$ ).
- SHA-2 também suporta saídas de 224, 256, 384 e 512 bits:
  - Facilita o uso de **SHA-2 ou SHA-3 como alternativas compatíveis**.
- SHA-3 também suporta *extendable-output function* (XOF):
  - **SHAKE128** e **SHAKE256**: produzem saídas de comprimento arbitrário.
  - Oferecem níveis de segurança de aproximadamente 128 e 256 bits.
  - SHA-1 e SHA-2 **não** possuem suporte a XOF.

# Visão de Alto Nível de SHA-3 e SHAKE

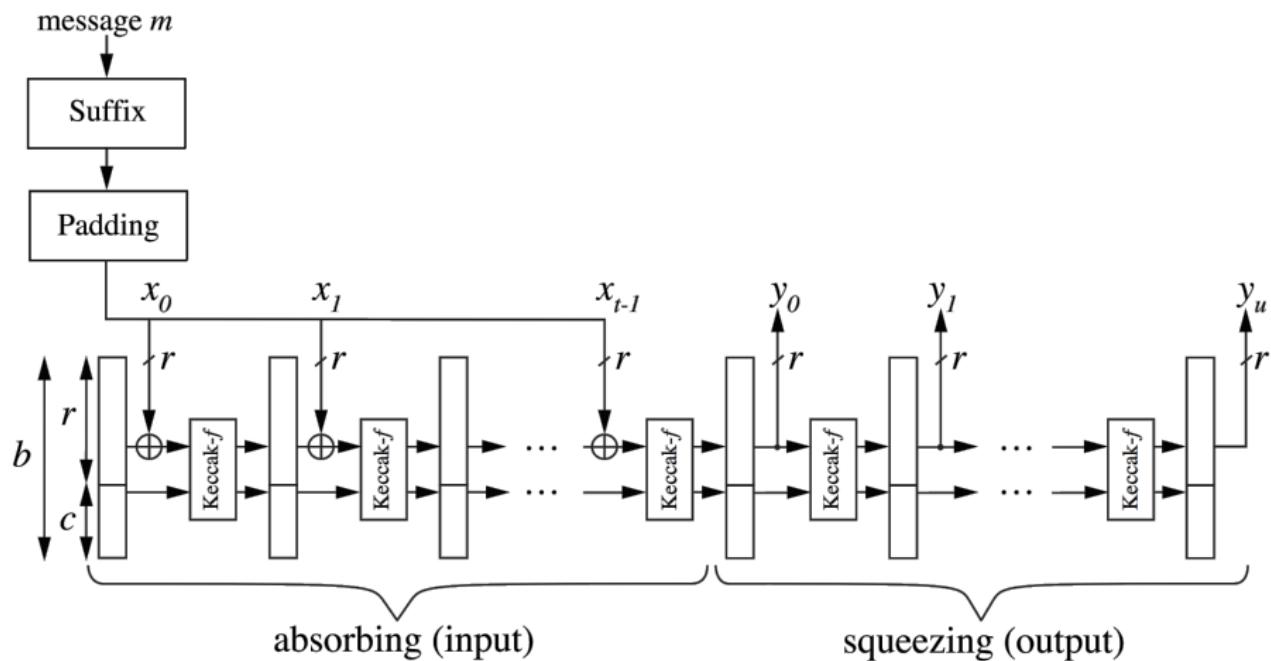
- **SHA-3** e suas variantes **SHAKE128/256** são baseadas no algoritmo Keccak.
- Keccak é o **núcleo criptográfico**: uma permutação utilizada no modo esponja.
- A construção **SHA-3/SHAKE** recebe uma mensagem  $m$  e:
  - gera um **digest de tamanho fixo**  $h(m)$  (caso SHA-3);
  - ou uma **saída de tamanho arbitrário**  $\text{string}(m)$  (caso SHAKE).
- Mesma base Keccak, **diferentes parâmetros de uso** (comprimento de saída, segurança efetiva).



# Keccak: Construção Esponja (Visão Geral)

- SHA-3 e as variantes **SHAKE128/256** são baseadas no algoritmo Keccak.
- SHA-3:
  - Recebe uma mensagem  $m$ .
  - Produz um **digest de tamanho fixo**  $h(m)$ .
- SHAKE128/256 (*Secure Hash Algorithm Keccak*):
  - São **funções de saída extensível (XOF)**.
  - Geram uma **string de bits de tamanho arbitrário**  $\text{string}(m)$ .
- Keccak é baseado em uma **construção esponja**:
  - Antes de tudo, há um **preprocessamento** de  $m$  (sufixo + padding, divisão em blocos).
  - Depois disso, a esponja possui duas fases:
    - ① **Fase de absorção**: processa os blocos de entrada  $x_i$ .
    - ② **Fase de espremer (squeezing)**: gera blocos de saída  $y_j$ .
- Keccak, por si só, permite gerar **quantos blocos de saída**  $y_j$  forem necessários:
  - Para **SHA-3**: apenas  $y_0$  é usado, e seus primeiros bits formam  $h(m)$ .
  - Para **SHAKE128/256**: usa-se a sequência  $y_0, y_1, \dots$  conforme a aplicação.

# Keccak: Construção Esponja



# Parâmetros da Esponja Keccak: $b$ , $r$ , $c$

- O “coração” é a função de permutação **Keccak-f**.
- Keccak-f opera sobre um **estado interno** de largura  $b$  bits:

$$b = r + c$$

onde:

- $b$  é a **largura do estado** (fixada em  $b = 1600$  bits).
- $r$  é a **taxa de bits (bit rate)**:
  - comprimento de cada bloco de mensagem  $x_i$ ;
- $c$  é a **capacidade**:
  - controla o **nível de segurança** da construção.
- Diferentes combinações de  $(r, c)$ , com  $b = 1600$  fixo, resultam em:
  - diferentes **desempenhos** (mais/menos bits absorvidos/expelidos por chamada de Keccak-f);

	function type	$b$ (state) [bits]	$r$ (rate) [bits]	$c$ (capacity) [bits]	security level [bits]	hash output [bits]
<b>SHA3-224</b>	hash	1600	1152	448	112	224
<b>SHA3-256</b>	hash	1600	1088	512	128	256
<b>SHA3-384</b>	hash	1600	832	768	192	384
<b>SHA3-512</b>	hash	1600	576	1024	256	512
<b>SHAKE128</b>	XOF	1600	1344	256	128	arbitrary
<b>SHAKE256</b>	XOF	1600	1088	512	256	arbitrary

# SHA-3 / SHAKE: Sufixo e Padding

- Antes do processamento de uma mensagem  $m$ , são aplicados:
  - um **sufixo** específico de SHA-3 ou SHAKE;
  - um **padding** definido na especificação de Keccak.
- O sufixo **não** faz parte da função Keccak em si, mas é exigido por SHA-3 e SHAKE128/256:
  - SHA-3:  $\text{suf} = 01$ ;
  - SHAKE128/256:  $\text{suf} = 1111$ .
- Esses sufixos diferentes garantem **separação de domínio**:
  - a mesma mensagem  $m$  não produz a mesma sequência de bits quando usada em SHA-3 e em SHAKE;
  - evita reutilização insegura do mesmo estado interno para funções com propósitos diferentes (hash fixo vs XOF/pseudorrandômico).
- Após anexar o sufixo, aplica-se o **multi-rate padding**:

$$\text{pad}(m, \text{suf}) = m \parallel \text{suf} \parallel 10^*1$$

onde:

- $10^*1$  é uma sequência que começa com 1, segue com o menor número possível de zeros e termina com 1;
- o objetivo é que o comprimento total seja um múltiplo de  $r$  bits.

# SHA-3 / SHAKE: Comprimento da Saída e Uso de $y_j$

- Após absorver todos os blocos de entrada  $x_i$ , Keccak entra na fase de **squeezing**, gerando blocos de saída  $y_0, y_1, \dots$ , cada um com  $r$  bits.
- **SHA-3:**
  - Apenas o **primeiro bloco**  $y_0$  é considerado.
  - $y_0$  tem  $r$  bits, mas o digest requerido é de 224, 256, 384 ou 512 bits.
  - Usa-se apenas os **bits mais significativos** de  $y_0$ ; os demais bits de  $y_0$  são descartados.
- **SHAKE128/256:**
  - Todos os  $r$  bits de  $y_0$  podem ser usados.
  - Se a aplicação necessita de mais que  $r$  bits:
    - aplica-se a permutação **Keccak-f** novamente ao estado;
    - gera-se  $y_1, y_2, \dots$  até alcançar o comprimento desejado.
  - Isso transforma SHAKE em uma **XOF**: função de saída extensível, útil como hash configurável, KDF ou PRNG.
- O tamanho de padding:
  - mínimo: sequência 11 (2 bits);
  - máximo: padrão 10...01 com comprimento  $r + 1$  bits.

# Função Keccak-f: Permutação Base de SHA-3

- Keccak-f é o núcleo de Keccak, logo de **SHA-3** e **SHAKE128/256**.
- É uma **permutação** sobre  $2^b$  estados:
  - Cada inteiro de  $b$  bits é mapeado *bijetivamente* em outro inteiro de  $b$  bits.
- O estado é visto como um arranjo 3D de bits:

$$b = 5 \times 5 \times w$$

- Coordenadas  $(x, y)$  formam uma **coluna**, e os  $w$  bits ao longo de  $z$  são chamados de **lane**.
- Para SHA-3/SHAKE:  $w = 64$  bits é mapeado para:  $b = 1600$  bits.
- O estado de 1600 bits encaixa bem em arquiteturas de 64 bits:
  - Pode ser armazenado como um vetor de 25 palavras de 64 bits.

# Keccak-f: Tamanhos de Estado e Estrutura de Rodadas

- Keccak-f suporta sete tamanhos de estado:

$$b = 25 \cdot 2^l, \quad l = 0, 1, \dots, 6$$

$$w = 2^l, \quad b = 5 \times 5 \times w$$

- Número de rodadas para cada  $b$ :

$$n_r = 12 + 2^l$$

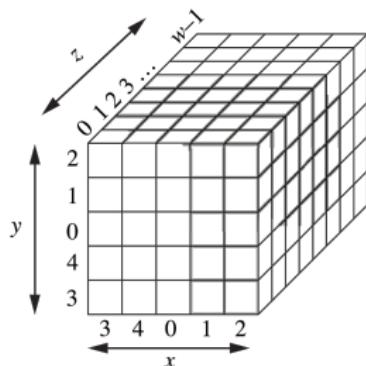
- Para SHA-3/SHAKE128/256:  $b = 1600$ ,  $w = 64$ ,  $l = 6$ , logo  $n_r = 24$ .
- Cada rodada de Keccak-f aplica, em sequência, cinco transformações lineares/não lineares:

$$\theta, \rho, \pi, \chi, \iota$$

- Todas as rodadas têm a mesma estrutura.
- Diferem apenas pela **constante de rodada**  $RC[i]$ , usada na etapa  $\iota$ .
- Essência: uma permutação altamente misturadora de 1600 bits, aplicada várias vezes, que confere à esponja Keccak suas propriedades de segurança.

# Visualização do estado interno

Visualização 3D do estado do Keccak, cada pequeno cubo representa um bit. Para o SHA-3 e o SHAKE128/256, a “profundidade” do arranjo ao longo do eixo  $z$  é  $w = 64$ .

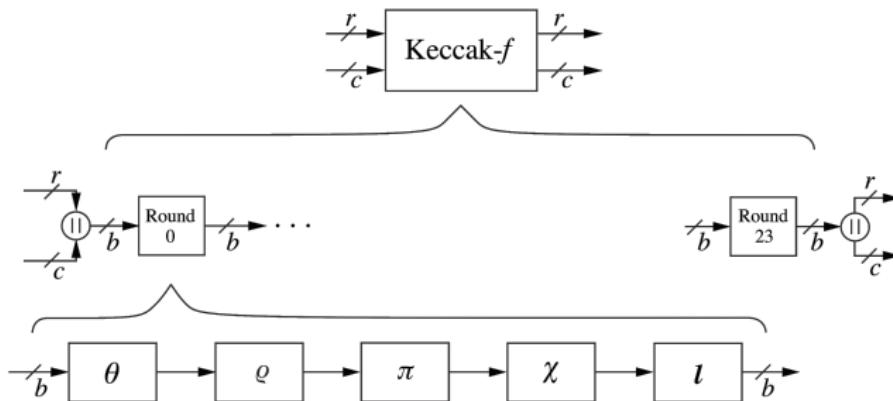


Os diferentes tamanhos de estado e o número de rodadas de Keccak-f; note que  $b = 1600$  e  $n_r = 24$  para o SHA-3 e o SHAKE128/256.

state $b$ [bits]	25	50	100	200	400	800	<b>1600</b>
number of rounds $n_r$	12	14	16	18	20	22	<b>24</b>
lane $w$ [bits]	1	2	4	8	16	32	<b>64</b>
$l$	0	1	2	3	4	5	6

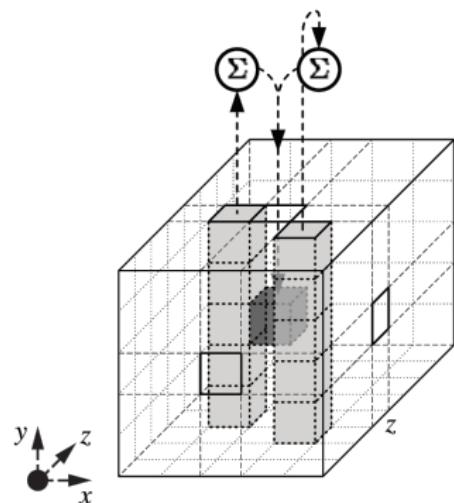
# Estrutura Interna da Função Keccak-f

- Cada chamada de Keccak-f aplica  $n_r = 24$  rodadas idênticas, e cada rodada é composta em série pelas etapas  $\theta$ ,  $\rho$ ,  $\pi$ ,  $\chi$  e  $\iota$ ; apenas a etapa  $\iota$  recebe uma constante de rodada  $RC[i]$  diferente em cada iteração.



## Etapa $\theta$ do Keccak-f

- Transformação linear de difusão do Keccak-f: cada bit do estado é substituído pelo XOR entre si e com outros 10 bits de sua vizinhança, espalhando a influência de cada coluna por todo o estado.
- Na implementação, calculam-se:
  - $C[x] = A[x, 0] \oplus A[x, 1] \oplus A[x, 3] \oplus A[x, 4]$
  - $D[x] = C[x - 1] \oplus \text{rot}(C[x + 1], 1)$
  - $A'[x, y] = A[x, y] \oplus D[x]$



## Etapa $\rho$ do Keccak-f

- O estado é visto como um arranjo  $5 \times 5$  de *lanes*  $A(x, y)$ , cada uma com  $w$  bits (para o SHA3 e SHAKE128/256  $w = 64$ ).
- Cada lane é submetida a uma **rotação circular de bits** com um deslocamento (offset) específico, que depende exclusivamente das coordenadas  $(x, y)$  dessa lane.
- Todos os deslocamentos são tomados módulo  $w$ , e foram escolhidos de forma a espalhar os bits ao longo da dimensão  $z$ , contribuindo para uma boa difusão em combinação com as demais etapas da permutação.
- A tabela abaixo mostra, para cada posição  $(x, y)$ , o número de bits de rotação aplicado na etapa  $\rho$ .

	$x = 3$	$x = 4$	$x = 0$	$x = 1$	$x = 2$
$y = 2$	153	231	3	10	171
$y = 1$	55	276	36	300	6
$y = 0$	28	91	0	1	190
$y = 4$	120	78	210	66	253
$y = 3$	21	136	105	45	15

## Etapa $\pi$ do Keccak-f

- A etapa  $\pi$  é uma **permutação das 25 lanes** do estado  $A(x, y)$  visto como uma matriz  $5 \times 5$ .
- Dado o novo arranjo  $A'$ , a regra de permutação é:

$$A'[x, y] = A[x + 3y, x], \quad x, y = 0, 1, 2, 3, 4$$

onde todas as coordenadas são computadas módulo 5.

- Nenhum bit é modificado, apenas **reordenado**:
  - a etapa  $\pi$  redistribui as lanes no plano  $(x, y)$ ;
  - prepara o estado para que as operações seguintes (especialmente a  $\chi$ ) combinem bits de posições que antes não interagiam.
- Exemplo: a posição  $A'[2, 3]$  será preenchida pela lane

$$A[2 + 3 \cdot 3, 2] = A[11, 2] = A[1, 2] \quad (\text{mód } 5),$$

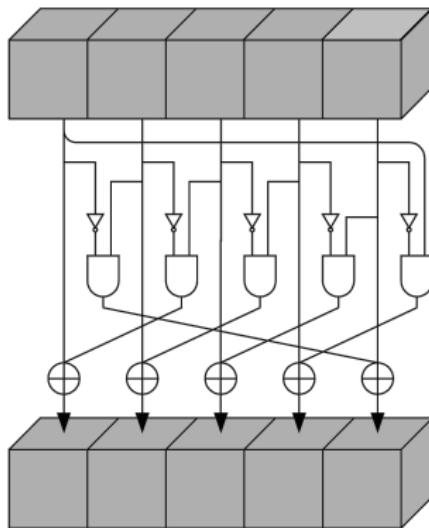
isto é, a lane originalmente em  $(1, 2)$  é movida para a borda inferior direita.

## Etapa $\chi$ do Keccak-f

- **Única operação não linear** dentro do Keccak-f.
- Opera por linhas de lanes: para cada coordenada  $(x, y)$  temos

$$A'[x, y] = A[x, y] \oplus (\overline{A[x+1, y]} \wedge A[x+2, y]), \quad x, y = 0, 1, 2, 3, 4$$

onde os índices são tomados módulo 5.



## Etapa $\iota$ do Keccak-f

- A etapa  $\iota$  adiciona uma **constante de rodada** de  $w$  bits à lane na posição  $(0, 0)$  do estado:

$$A'[0, 0] = A[0, 0] \oplus RC[i].$$

- A constante  $RC[i]$  depende da rodada  $i$ :
  - o número de rodadas  $n_r$  varia com o parâmetro  $b$ ;
  - para SHA-3 e SHAKE128/256 temos  $n_r = 24$  rodadas, logo  $RC[0], \dots, RC[23]$ .
- Cada  $RC[i]$  é, em essência, um vetor quase todo nulo, com bits pseudoaleatórios em posições específicas (por exemplo 1, 2, 3, 7, 15, 31, 63), gerados por um LFSR de grau 8.
- A função da etapa  $\iota$  é **quebrar simetrias** entre rodadas: ela insere dependência explícita do índice de rodada na permutação Keccak-f, reforçando a segurança contra ataques estruturais.

# Lições Aprendidas sobre Funções Hash

- Funções hash são **sem chave** (keyless) e têm muitas aplicações em sistemas de segurança modernos, como **assinaturas digitais** e **MACs**.
- As três propriedades de segurança fundamentais são:
  - **one-wayness** (pré-imagem difícil),
  - **segunda pré-imagem** difícil,
  - **resistência a colisões**.
- Para resistir a ataques de colisão, a saída de uma função hash deve ter, em geral, **256 bits ou mais**.
- **SHA-2** e **SHA-3** são considerados altamente seguros atualmente; não há ataques práticos conhecidos. Já o **SHA-1** é considerado inseguro.
- Além de algoritmos dedicados (como SHA-2 e SHA-3), é possível construir funções hash a partir de **cifras de bloco**.
- **SHA-3** baseia-se em uma **construção esponja**, o que o torna, internamente, bem diferente de SHA-1 e SHA-2.
- Em **hardware**, SHA-3 tende a ser mais eficiente, sendo adequado para aplicações móveis e embarcadas; em **software**, SHA-2 costuma ser mais rápido do que SHA-3.